**Generic Workflow and DLT (POC)**

Last Updated: 02/10/2025  
Ref:

|  |  |
| --- | --- |
| 252559 (User Story) | Explore generic pipeline & limitations |

**Objective:**

The goal is to consolidate multiple pipelines with varying parameters into a **single generic pipeline** that can dynamically adapt to different inputs. This pipeline should be callable via REST API using tools like Postman.

A diagram of a computer program

Description automatically generated

**Summary:**  
Making the framework dynamic is feasible up to the raw layer. However, for the bronze layer (DLT), it is not currently possible, as we observed that in all the options we explored, tables are getting dropped when required. We can connect with Databricks to explore potential solutions for dynamically creating the DLT without tables being dropped.

**Option 1: Using a Non-DLT Bronze Layer**

* Given the challenges in making DLT fully dynamic, adopting a non-DLT approach for the bronze layer provides greater flexibility and control over data ingestion and processing.
* Implementing a robust audit table and custom logging mechanism ensures comprehensive tracking of different data sources, improving data transparency and traceability.

**Option 2: Common Up to Raw, Varies by Source at Bronze**

* We can make the raw layer generic and, since API calling is possible, we can dynamically create the DLT pipeline using the API.
* Implementing a robust audit table and custom logging mechanism ensures comprehensive tracking of different data sources, improving data transparency and traceability.

**Options explored (Implementation Details)**

We will implement a single workflow and a single DLT pipeline to cater to all use cases (e.g., Boost Retail, PTS Jet, etc.).

The workflow's structure and layout will remain consistent. When invoking the workflow through a POST API, notebook parameters can be specified, which will then be applied to the task parameters. Additionally, a PUT API will be used concurrently to modify the existing DLT, ensuring it can be effectively utilized across all use cases.

**Notebook name:** Workflow and DLT API Calling

**Notebook Path:** /Workspace/Shared/DNAP-V2.3/249714-Table-Env-Mapping-Final/Workflow and DLT API Calling

**Workflow Name:** M3\_Medallion\_Framework\_Generic

**Key Use Cases & Findings in DLT Implementation**

**1. API-Based Workflow Execution & DLT Update:**

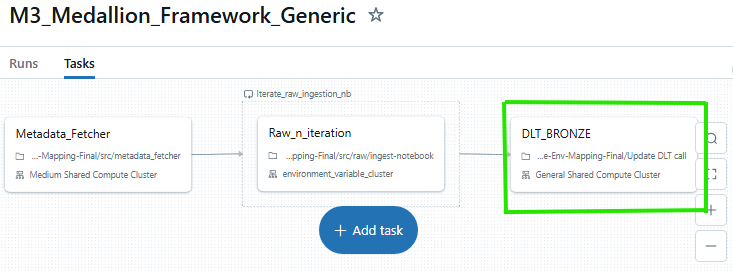
* The workflow and DLT pipeline can be successfully invoked via API. This has been confirmed through testing in both Databricks Workspace and Postman.
* For parameters, it’s recommended to use notebook parameters(widgets). Task values can be directly accessed but it creates a tight coupling that wouldn’t be desired.
* The DLT pipeline does not accept parameters when called via API. To handle different use cases, the configuration path must be updated using a PUT API.
* The entire workflow executes successfully when initiated.
* When the workflow is executed for a second time with a different configuration path, it is observed that tables from the previous configuration path are dropped.
* It has been concluded that changing (read, using PUT or modifying the DLT pipeline, not just passing new values) the configuration path does not allow DLT to refresh the previous tables. Since these tables are no longer part of the pipeline, they are deleted from the catalog.

**2. Direct Bronze Code Execution in Workflow:**

* Calling the Bronze DLT notebook at the workflow task level is not possible because the DLT pipeline must be associated with it for execution.

**3. Bronze Code Execution via API in Workflow:**

* In this approach, we call a notebook containing two APIs: one for updating the DLT pipeline and another for starting it.

****

* While the pipeline executes successfully, we observe the same behaviour as in the **API-Based Workflow Execution & DLT Update** use case, where tables are getting dropped.

**4. Production DLT Creation:**

* While invoking DLT, we are switching from development mode to production to determine if this prevents the table from being dropped.

**A screen shot of a computer program

Description automatically generated**

* While the pipeline executes successfully, we observe the same behaviour as in the **API-Based Workflow Execution & DLT Update** use case, where tables are getting dropped.

**5. Adding pipelines.reset.allowed property:**

* While creating the bronze layer table, we are setting the pipelines.reset.allowed property and then creating the table.

A screenshot of a computer code

Description automatically generated

* While the pipeline executes successfully, we observe the same behaviour as in the **API-Based Workflow Execution & DLT Update** use case, where tables are getting dropped because this property controls whether a full refresh is allowed for this table.

**6. Storing All JSON Files in a Single Location:**

Currently, we maintain separate folders for each data source. Instead, we can consolidate all JSON files in a single location, ensuring that all tables are associated with a single DLT pipeline. However, this approach requires triggering tables for all data sources, even if not needed at a given time, as failing to do so even once would result in table drops.

**Conclusion:**

The objective of creating a dynamic, API-driven pipeline was partially achieved. While the workflow and DLT pipeline can be successfully invoked via API and notebook parameters can be dynamically passed, the current approach of changing configuration paths leads to the deletion of previous tables.

A screenshot of a computer

Description automatically generated